# Teaching Software Development to non-Software Engineering Students

John Colvin
University College Worcester
Henwick Grove, Worcester, United Kingdom

*The paper argues that although the object-oriented programming (OOP) paradigm is appropriate for students taking programming modules on Higher Education (HE) software engineering course, this paradigm is not as relevant for students from other courses who study programming modules. It is also asserts that adopting another paradigm when teaching programming to non-software engineering students need not prevent the encouragement of good software engineering practices*

*The paper discusses the software development model, procedures, techniques and programming language that the author requires non-software engineering students to employ when developing their software. This discussion also includes consideration of implementation issues in an educational context.*

*The paper concludes that his alternative approach has been successfully implemented, that it requires the student to adopt a rigorous approach to development and that it encourages best software engineering practices. The conclusions also note that delivering this alternative offers the opportunity to include good educational practice, such as role-play.*

## 1.0 Introduction

Computer programming is taught to Higher Education (HE) students on a wide range of courses. Although some of these students are enrolled on Software Engineering (SE) courses, others are enrolled on computing courses that do not place an emphasis on software engineering, for example Multi-media, or on non-computing courses, such as Business.

The author's early experience was that teaching non-SE students in the same way as students on a SE course was not successful. This experience encouraged him to adapt his teaching of programming to non-SE students. Initially this involved ensuring that the software created by students was relevant to their curriculum area. The author's current approach focuses on students creating small-sized applications using an iterative software development model and an end-user programming language.

The paper concentrates on the author's current approach. It explains the rationale behind this approach to teaching programming and discusses how the approach has been delivered in an educational environment. To facilitate detailed reflection, the software development model, procedures and techniques that students are required to use in their programming are articulated. Additionally, to reassure the reader that good software engineering practices have not been ignored, instances where they are encouraged are highlighted.

## 2.0 Argument for an Alternative Paradigm

The object-oriented programming (OOP) paradigm dominates HE and the experience of some is that it is an ideal first paradigm for students to learn [30]. It is claimed that students experience difficulty in moving to the OOP paradigm having experienced another first paradigm [17]. Other reasons in favour of choosing the OOP are articulated in Decker et al [10]. This adoption has been accompanied in the UK by the predominant use of Java [8]. These arguments for adopting the OOP as a first paradigm are persuasive when teaching potential software engineers, as their future employment in this field is likely to require significant use of the OOP paradigm.

However, many undergraduate programming modules are delivered to students who do not aspire to employment as software engineers. These non-SE students might be enrolled on computing courses that do not place an emphasis on software engineering (e.g. Multi-media) or on non-computing courses (e.g. Business Studies). In future employment, non-SE students are likely to apply their acquired programming skills only as a subsidiary part of their job. These skills could well be practiced using an end-user programming language, possibly embedded in an applications package e.g. Lingo in Macromedia Director, VBA in Microsoft Office, VBScript in web page design. End-user development is now widespread [6] resulting, the author suggests, from the failure of software resources to do fulfill requirements [26]. Generally, such development does not require the use of an OOP language. Given that these non-SE are not likely to use the OOP paradigm in future employment and that they will inevitably study less programming modules (than SE students) tends to suggest that the OOP paradigm might not be an appropriate choice. As programming modules are improved by an appropriate choice of language [26], the author uses different languages when teaching different groups of non-SE students (e.g. Lingo for Multi-media students). This choice will hopefully deliver the motivation that Jenkins et al [15] considers as being crucial.

## 3.0 Need to Emphasise Good Software Engineering Practice

The end-user development suggested generally entails using a visual rather than an OOP language. However, not selecting the OOP paradigm should not prevent our encouraging good software engineering practices, as these practices can be applied in visual languages to improve the quality of a product [28]. The fact that, when employed, these students (unlike non-SE students) are likely to be practicing their skills outside the control of a software development environment further justifies our encouraging good software engineering practices.

The author's own experience is that, when left to their own devices, many non-SE students, who are developing a small-sized high-level program, adopt a prototyping approach that does not systematically ensure planning, robust design, testing, completed documentation and client evaluation for each phase of development. Indeed, Weaver [32] articulates a similar experience with undergraduate students and Avison [1] warns of developers neglecting documentation and not carrying out thorough analysis and design when using prototyping. Perhaps this is not surprising, although there are general guidelines on iterative software development in well respected software engineering texts such as Pressman [21] and Sommerville [27] and there are also some specific guidelines on particular aspects of Event Driven Programming development, e.g. Philip [20], the author is not aware of the existence of aggregate detailed guidelines for the type of development proposed.

Advice exists on the critical software engineering practices that are essential if major software engineering development problems are to be avoided [25] [33].

## 4.0 Choice of Software Development Model

As with modules aimed at SE students, it is not just about teaching students how to code, nor is it about teaching requirements elicitation, design, coding and testing as separate activities. However, it is necessary to be vigilant as the author's experience is that some non-SE students find programming modules challenging and request an emphasis on coding. Students need a framework (e.g. software development model) to guide and rationalise these activities.

The inflexible nature of linear development [27] [29] suggests that this developmental approach is unsuitable. Generally, programming students are unlikely to be experienced in the development environment, in the application area and in the development process. However, when non-SE students subsequently program in employment it could well be on an occasional basis resulting in their being rusty and not fully au fait with the development environment. The author's experience is that typical clients for a small-sized project possess only a general overview of their requirements and the lack of experience of such clients in participating in software projects will not alert them to the consequences of signing off requirements too early.

The author requires non-SE students to use an iterative software development model; these iterative development models are increasingly popular for the development of small-sized systems [27]. Students

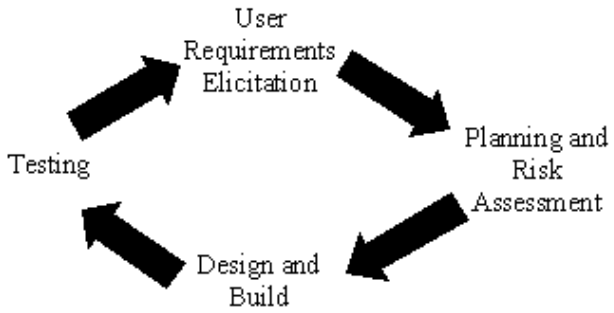


**Fig. 1**

are likely to be relatively inexperienced in many aspects of software development and so adopting an iterative approach will not only prevent their inexperience jeopardising the project, but also offer them the opportunity to practice their skills. Developing iteratively can also overcome the difficulties of requirements elicitation identified in Pressman [21] by better matching client requirements and identifying additional functionality through repeated client evaluation of the product. Finally the benefits arising from user participation will accrue following an increased involvement of the client [1].

The software development model adopted (Fig. 1) is iterative and the division of each iteration is not dissimilar to that proposed by other authors (e.g. Boehm [2]). Notionally each iteration is linear, however this should not prevent the student from interrupting or adapting an iteration if circumstances dictate.

# 5.0 Software Development Activities

The software development model must be implemented in an educational environment and the associated software development activities phased in over an undergraduate course. Irrespective of the relevance of a model, some students are inclined to adopt a least effort approach to development, similar in many ways to the "least effort research model" proposed by Chrzastowski [7]. This approach will not afford students the opportunity to practice and appreciate the procedures and techniques considered essential to their education and careers. This lack of exposure is compounded by the relatively few programming modules that non-SE students will take, compared to SE students. The author exploits the focus that students place on assessment [11] to encourage full engagement with the techniques and procedures suggested.

Often the paper considers activities in an assessment context. Programming assessments can be categorised as having or not having a so-called live client. Larger student projects (e.g. a final year student project) usually involve a live client, whereas more usually students exercises or assignments are based on a specification designed by the lecturer.

## 5.1 User Requirements Elicitation

Requirements elicitation has been identified as the most difficult aspect of development [5] and it is surprising that Myers [19] reports that traditional Computer Science students have little experience of requirements analysis when undertaking software projects. Pressman [21] advises a developer to elicit requirements in an organised way to overcome the problems suggested in Christel [9] and so the author requires non-SE students and SE students adopt a similar approach for small-sized development.

The small size of such projects suggests that the primary elicitation method would be individual face-to-face meetings with the client. Pressman [21] offers useful advice on conducting such meeting. Other techniques such as questionnaires, group meetings and focus groups are more applicable to larger projects and so might be included in a module aimed at SE students.

Immediately following meetings, the student creates or updates Use Case Diagrams to illustrate the overall context of the system and Use Case Scenarios to illustrate the system's functional requirements (the 'What'). Scott [24] reassures us that developers do not experience difficulty in creating and refining Use Cases as a product evolves. Use Cases are key to UML [4] and are described in numerous texts including Booch et al [4] and Scott [24]. The construction of Use Case documentation will require analysis of requirements by the student.

When a live client is involved, realistic iterative requirements elicitation can take place. Furthermore the constraints of predetermined submission dates and student workloads will simulate work pressures to complete within a given timescale. In the more usual assessment situation when a live client is not

involved, role-play [3] can be used with the lecturer playing the role of the client. The large groups that are now typically taught in HE preclude the use of individual requirements elicitation for each student and suggest the need for a collective solution. The use of role-play affords the additional advantages of allowing the lecturer to offer feedback on interview technique to students and for the lecturer to receive feedback on their teaching interview techniques.

## 5.2 Planning / Risk Management

Students often have limited experience of projects. They do not appreciate the need for planning and take more time than anticipated to complete tasks [32]. Non-SE students are required, as are my SE students, to project plan and undertake risk analysis. Carrying out these activities during each development iteration is important as it provides students with useful experience and completing these activities is a critical factor in successful student projects [32]. Interestingly, formal risk management has been identified as the most important software engineering best practice for industry [33].

To ensure that students do not complete these activities retrospectively a number of strategies can be adopted. One strategy is to require fragmented submission of documentation at the end of each iteration as suggested in Jackson et al [14] on the grounds that it emulates processes used in industry. Another strategy is to acknowledge that these activities are very difficult and place less emphasis on penalising students for making errors in planning and risk assessment, but more emphasis on rewarding them for analysing their decision making in a reflective diary.

A fragment from risk analysis documentation that a student might submit is presented in Table 1.

**Table 1**

| **Risk Analysis** | | **Pro-forma Header** (Name, Project ID, Document ID etc.) | |
|---|---|---|---|
| **Risk** | **Likelihood** | **Impact** | **Action** |
| Conflicting coursework deadlines | H | M | Investigate coursework deadlines of other modules |
| PC down time | L | H | Take regular back-ups<br>Check licensing arrangements to use compiler on another PC |

## 5.3 Design

Clearly not adopting an OOP paradigm will entail the use of some different techniques than when teaching OOP to SE students. The visual nature of these applications requires the student to design both the interface and the underpinning functionality caused as a result of the user's interaction with the interface.

**5.3.1 User System Interface Design:** A survey in Langay et al [16] indicated that storyboards were popular for early concept sketches. Although this technique might be appropriate for simple applications, more complex interfaces will demand a technique that will enable a rich and concise expression of interface functionality. Bubble State Transition Diagram (STD) technique can be used, but again this technique is not suitable for complex systems [18]. The use of Statecharts, developed by David Harel [12], is an alternative that embodies extensions to STD to enable the concise expression of a complex interface specification. The specific application of the Statecharts technique for User Interface design is developed in Horrocks [13]. This technique is widely used and is part of Unified Modelling Language UML [4].

**5.3.2 Functional Design:** The Event Driven paradigm that obliges the student to initially decompose the problem into sub problems associated with event procedures [20]. Events are specified in the statechart created in USI design (see above) and this statechart is used, in conjunction with Use Case documentation, to validate design decomposition. As previously mentioned, Use Case scenarios specify the so-called

'What' to which must be added the so-called 'How'. State Event Logic (SEL) Charts, adapted from Event-action tables [13] are used to specify the underpinning functionality (not to be confused with USI functionality). For each state/event situation, a SEL Chart details the pseudo-code design to specify the 'How' and subsequent state. Reservations have been expressed about the use of pseudo-code [25], however the author's experience is that students find this technique both useful and intuitive. Table 2 illustrates the headers of a skeleton SEL Chart

**Table 2**

| **SEL Chart** | | **Pro-forma Header** (Name, Project ID, Document ID etc.) | |
|---|---|---|---|
| **State** | **Event** | **Pseudo Code** | **State** |
| | | | |

## 5.4 Coding

As with my SE students, this is carried out against coding standards. These prescribe scope of variables, reusability requirements for procedures, nomenclature, etc. The author's own preference is to prescribe such standards (see 5.6.). An alternative approach that requires students to develop their own standards has much merit but might be better suited to a separate exercise.

## 5.5 Testing

As professional software engineers frequently view testing as an afterthought [21], we should not be surprised by the author's experience of non-SE students, and SE students, allocating insufficient resources to testing. Indeed, [22] reports a similar experience. This attitude might be exaggerated by an incorrect perception that each product release is only a prototype to be discarded and as such not worthy of testing. It is therefore necessary to explain and emphasise the need for testing. This emphasis has been highlighted by including a separate stage into the development model - it is more customary for iterative development models to include testing in the design/build stage.

Testing occurs each iteration, in line with conventional wisdom that bugs should be isolated and resolved as soon as possible [31]. This expeditious correcting should not only yield resource savings, but also negate the possible loss of client confidence that results from using a product that includes bugs. Testing is most effective when carried out independently [21]. In order to support students in the development process and also to expose them to the professional practice of third party testing and peer-review inspections students are required to collaborate with other students during most aspects of this phase. This collaboration should also offer potential benefits of learning from each other [23].

Withers [33] recommends that testing should not be confined to program code and so the following have been incorporated - functionality/structural testing, usability testing, code design inspection and user evaluation. Additionally there are two key aspects of testing that are not carried out during this phase of the cycle. Firstly, requirements testing is performed during and immediately after requirements elicitation are documented in order to avoid the extra remedial effort required when requirements errors are identified after coding. Secondly, design testing is carried out before coding, in line with the advice to "Design Twice and Code Once" [25].

## 5.6 Software Configuration Management

Iterative development encourages change and change causes confusion amongst developers [21], coupling this with the inclination of students to see documentation as almost irrelevant [22] suggests the need to require students to undertake configuration management. Not surprisingly, configuration management is identified as a critical software engineering practice [25]. This

activity not only supports students in the successful completion of their work but also allow them to practice industry relevant techniques. Students experience difficulty in identifying good documentation [22] and so it is reasonable to anticipate that they will also experience difficulty with the broader discipline of software configuration management. The author therefore specifies software configuration procedures, conventions and standard documentation for students.

# 6.0 Conclusions

Although the argument for adopting the OOP paradigm for SE students is persuasive, this approach is not as relevant when teaching programming to students who are enrolled on courses that do not place an emphasis on software engineering. For these students it is more appropriate to concentrate on developing small-sized systems using an end-user programming language. Using such a language negates the need for the OOP paradigm.

This alternative approach has been successfully implemented using an iterative software development model that both requires students to adopt a rigorous approach to software development and also includes best software engineering practices in activities such as configuration management, planning, risk management, requirements elicitation, inspections and testing. Strategies, such as exploiting the focus that students place on assessment, are employed to ensure that students fully engage with the software development procedures and techniques involved.

Delivering this alternative to students in an HE context offers the opportunity to include good educational practice, such as role-play and working with others.

# 7. References


[1] Avison, D., and Fitzgerald, G., Information Systems Development, McGraw-Hill, 2003.

[2] Boehm, B., "A spiral model for Software Development and Enhancement" Computer, 21(5), 1988, pp.61-72.

[3] Bolton, G., and Heathcote, D., So You Want to Use Role-Play?, Trentham Books, 1999.

[4] Booch, G., Rumbaugh, J., and Jacobson, I., The Unified Modelling Language User Guide, Addison Wesley, 1999.

[5] Brooks, F., "No Silver Bullet Essence and Accidents of Software Engineering", Computer, 20(4), 1986, pp.10-20.

[6] Burnett, M., Cook,C., and Rothermel, G., "End-User Software Engineering", Comms of the ACM, 47(9), 2004, pp.53-58.

[7] Chrzastowski, T., "E-journal access", Library Computing, 18(4), 1999, pp. 317-322.

[8] Culwin, F., "Object Imperatives!", SIGCSE '99, 1999, pp. 31-36.

[9] Christel, M. and Kang, K , "Issues in Requirements Elicitation", Software Engineering Institute Technical Report CMU/SEI-92-TR-012, 1992.

[10] Decker, R., and Hirshfield, S., "The Top Reasons Why Object-Oriented Programming Can't Be Taught in CS1", SIGCSE '94-, 1994, pp. 51-55.

[11] Gibbs, G., and Habeshaw, T. Preparing to teach: An introduction to effective teaching in higher education, Technical and Educational Services Ltd, 1989.

[12] Harel, D., "Statecharts: a ", Science of Computer Programming, 8(3), 1987, pp. 231-274.

[13] Horrocks, I., Constructing the User Interface with Statecharts, Addison-Wesley, 1999.

[14] Jackson, U., Manaris, B., and McCauley, R., "Startegies for Effective Integration of Software Engineering Concepts and Techniques into the Undergraduate Computer Science Curriculum", SIGCSE '97, 1997, pp. 360-364.

[15] Jenkins, T., and Davy, J., "Diversity and Motivation in Introductory Programming", ITALICS, 2002, 1(1), available online at www.ics.ltsn.ac.uk/pub/italics/issue1/tjenkins/003.html.

[16] Langay, J., and Myers, B., "Sketching Storyboards to Illustrate Interface Behaviors", CHI '96, 1996, pp.193-194.

[17] Marion, W., "CS1: What Should We Be Teaching?", SIGCSE Bulletin, 31(4), 1999.

[18] Martin, J., and McClure, C., Diagramming Techniques for Analysts and Programmers, Prentice-Hall, 1985.

[19] Myers, J., "Software Engineering Through A Traditional CS Curriculum", JCSC, 16(2), 2001, pp.31-41.
[20] Philip, G., "Software design guidelines for event-driven programming", The Journal of Systems and Software, 41, 1998, pp.79-91.
[21] Pressman, R. S., Software Engineering A Practitioner's Approach, McGraw-Hill, 2000.
[22] Robergé, J., and Suriano, C., "Using Laborities to Teach Software Engineering Principles in the Introductory Computer Science Curriculum", SIGCSE 94, 1994, pp.106-110.
[23] Rowntree, D., "Teaching and learning online", British Journal of Educational Tech, 26(3),1995, pp.205-215.
[24] Scott, K., UML Explained, Addison-Wesley, 2001.
[25] Software Program Managers Network, 16 Critical Software practices for Performance-based Management, 1999, available online at www.spmn.com/critical_software practices.html.
[26] Soloway, E., "Should We Teach Students To Program?", Communications of the ACM, 36(10), 1993, pp.21-24.
[27] Sommerville, I. Software Engineering (7th ed.), Addison-Wesley, 1995.
[28] Sparkman, T.G., "Lessons Learned Applying Software Engineering Principles to Visual Programming", Procs of 23rd Annual International Computer Software & Applications Conference, Los Alamitos, , 1999, pp.416-421.
[29] Stillier, E. & LeBlanc, C., Project-Based Software Engineering, Addison Wesley, 2002.
[30] van Roy, P., Armstrong, J., Flatt, M., and Magnusson, B., "The Role of Language Paradigms in Teaching Programming", SIGCSE '03, 2003, pp. 269-270.
[31] van Vliet, H., Software Engineering, Wiley, 2000
[32] Weaver, P., Success in Your Project, Prentice Hall, 2004.
[33] Withers, D., "Software Engineering Best Practice Applied To The Modelling Process", Proceedings of the Winter Simulation Conference, Orlando, 2000, pp. 432-439.